electronics

MDPI

Review

AI-Assisted Programming Tasks Using Code Embeddings and Transformers

Sotiris Kotsiantis [1,*], Vassilios Verykios [2] and Manolis Tzagarakis [3]

[1] Department of Mathematics, University of Patras, 265 04 Patras, Greece
[2] School of Science and Technology, Hellenic Open University, 263 35 Patras, Greece; verykios@eap.gr
[3] Department of Economics, University of Patras, 265 04 Patras, Greece; tzagara@upatras.gr
* Correspondence: sotos@math.upatras.gr

Abstract: This review article provides an in-depth analysis of the growing field of AI-assisted programming tasks, specifically focusing on the use of code embeddings and transformers. With the increasing complexity and scale of software development, traditional programming methods are becoming more time-consuming and error-prone. As a result, researchers have turned to the application of artificial intelligence to assist with various programming tasks, including code completion, bug detection, and code summarization. The utilization of artificial intelligence for programming tasks has garnered significant attention in recent times, with numerous approaches adopting code embeddings or transformer technologies as their foundation. While these technologies are popular in this field today, a rigorous discussion, analysis, and comparison of their abilities to cover AI-assisted programming tasks is still lacking. This article discusses the role of code embeddings and transformers in enhancing the performance of AI-assisted programming tasks, highlighting their capabilities, limitations, and future potential in an attempt to outline a future roadmap for these specific technologies.

Keywords: AI-assisted programming; code embeddings; transformers

Citation: Kotsiantis, S.; Verykios, V.; Tzagarakis, M. AI-Assisted Programming Tasks Using Code Embeddings and Transformers. *Electronics* **2024**, *13*, 767. https://doi.org/10.3390/electronics13040767

Academic Editors: Galina Ilieva and George A. Tsihrintzis

Received: 19 January 2024
Revised: 8 February 2024
Accepted: 13 February 2024
Published: 15 February 2024

1. Introduction

AI-assisted programming or development is defined as the utilization of machine learning models trained on the vast amount of available source code. Its purpose is to support various aspects of programming and, more broadly, software engineering implementation tasks. According to the software naturalness conjecture [1], which posits that source code, like natural language, is often repetitive and predictable, this technology has become integrated into popular integrated development environments (IDEs) and gained widespread popularity among developers [2]. Noteworthy applications such as IntelliCode [3], Github Copilot [4,5], Codex [6], and DeepMind AlphaCode [7] exemplify AI-assisted programming tools accessible to the public.

The impact of AI on software development tasks is expected to enhance precision, speed, and efficiency [8]. These benefits extend beyond professional programmers to include novice programmers [9], with ongoing studies exploring the potential of AI in various fields. Research reports highlight the sensitivity of these tools to the specific tasks they support.

Despite generating code, AI-assisted programming tools may produce complex and error-prone code [10]. In the domains of data science and data analysis, these tools contribute positively to addressing challenging problems [11,12]. For novice programmers in educational settings, AI-assisted programming tools increase project completion rates and grades [13,14]. Nevertheless, novices may encounter difficulties in comprehending and utilizing these tools proficiently [15].

Code embeddings and transformers represent popular approaches to AI-assisted programming, significantly impacting software engineering by improving task performance

and efficiency. These techniques reduce manual effort in coding, debugging, and maintenance, thereby decreasing overall development time and costs. Furthermore, they enable cross-language development, allowing seamless work with multiple programming languages.

Code embedding [16] is a machine learning technique representing code as dense vectors in a continuous vector space. Unlike traditional methods that treat code as sequences, code embedding captures semantic relationships between code snippets by training a neural network to learn fixed-size vector representations. These embeddings find application in various software engineering tasks, such as code completion, correction, summarization, and search [17].

Researchers like Azcona et al. [18] propose using embeddings to profile individual Computer Science students, analyzing Python source code submissions to predict code correctness. Similarly, Ding et al. [19] introduce GraphCodeVec, employing graph convolutional networks to generate generalizable and task-agnostic code embeddings, demonstrating superior performance in multiple tasks.

Transformers [20], a type of neural network utilizing attention mechanisms for sequential data processing, stand out from traditional recurrent neural networks. Transformers can handle parallel input and self-attention mechanisms, processing a sequence of tokens by attending to all input tokens simultaneously. In contrast, code embeddings use traditional deep learning models like recurrent neural networks (RNNs) and convolutional neural networks (CNNs). Transformers, trained through pre-training and fine-tuning, can handle variable input lengths, whereas code embeddings require fixed input lengths.

For code-related tasks, input code snippets feed into the transformer model, which employs self-attention mechanisms to capture contextual relationships within the code. The transformed representations generated using the model find application in downstream software engineering tasks, such as code generation, summarization, translation, or identifying patterns and anomalies in code. Chirkova and Troshin [21] demonstrated improved performance with syntax-capture modifications in transformer models.

In Figure 1, the timeline showcases the evolution of AI-assisted programming tasks utilizing code embeddings and transformers from their early stages of experimentation to their integration as indispensable tools in modern software development workflows.

Figure 1. Timeline of the main contributions of AI-assisted programming tasks.

In summary, this paper details code embeddings and transformers, discussing their characteristics. It explores existing AI-supported programming approaches, contextualizing their support for various tasks. The paper concludes with insights and acknowledged limitations.

2. Code Embeddings and Transformers

Code embeddings, also referred to as source code embeddings or program embeddings, have garnered significant attention in the realms of natural language processing (NLP) and code generation. These techniques, categorized as representation learning, aim to encode both syntactic and semantic information from source code into a lower-dimensional vector space. While code embeddings have demonstrated promising results in various NLP tasks such as code similarity, bug detection, and code completion, recent advancements in transformer models have led to a shift in focus towards using transformers for code representation and generation tasks. This section delves into the concept of code embeddings and their relationship with transformers.

In contrast to conventional approaches relying on static program analysis techniques, code embeddings offer more effective ways to represent and analyze source code [16]. These embeddings employ neural networks to create a semantic representation of code sequences, learning from a substantial code corpus. The specific architectures and training techniques employed enable these networks to capture inherent patterns and relationships between code elements. Consequently, code embeddings encode both structural and lexical characteristics of source code, presenting a comprehensive representation applicable to diverse downstream tasks [19].

The process of creating code embeddings involves several steps. Initially, the source code undergoes tokenization, breaking it down into a sequence of tokens, which can be characters, words, or syntactic constructs of the programming language. Subsequently, this token sequence is fed into a neural network, and trained to generate a vector representation for each token. This process is repeated for all code sequences in the training data, and the resulting vectors are stored in an embedding matrix. The embedding matrix is considered a form of "learned parameters" in the neural network architecture. In detail, at the beginning of training, the embedding matrix is initialized with random values or pre-trained word embeddings. During the forward pass of the neural network, the input tokens (words) are represented as one-hot vectors or integer indices that correspond to their positions in the vocabulary. These indices are then used to index into the embedding matrix, retrieving the dense vector representations (embeddings) of the input tokens. During training, the values of the embedding matrix are adjusted via backpropagation and gradient descent. The objective is to learn meaningful representations of words that capture semantic relationships and contextual information from the training data. This process involves updating the parameters of the embedding matrix to minimize the loss function of the neural network.

One key technique employed in training code embeddings is the use of skip-gram models [22]. Initially developed for natural language processing tasks, skip-gram models are adapted for code embeddings to learn semantic relationships between code tokens based on their contextual surroundings. This enables the model to capture both syntactic and semantic aspects, yielding a more holistic representation.

In the realm of program synthesis, code embeddings find application in generating code from natural language descriptions [23]. This entails training the model to comprehend the relationship between natural language descriptions and code sequences, facilitating the generation of code aligned with the provided description. This application extends to automatic code documentation and the development of programming tools for non-technical users.

To summarize, the mathematical representation of code embeddings involves the following:

- Tokenizing the code snippet S to obtain a sequence of tokens (t_1,t_2,. . .,t_n).
- Obtaining the embedding E(t_i) for each token t_i.

- Combining the token embeddings to obtain the code embedding C, for example, by averaging.

Rabin et al. [17] evaluated the use of code2vec embeddings compared to handcrafted features for machine learning tasks, finding that code2vec embeddings offered even information gains distribution and exhibited resilience to dimension removal compared to handcrafted feature vectors.

Sikka et al. [24] introduced a machine learning problem related to estimating the time complexity of programming code. Comparing feature engineering and code embeddings, both methods performed well, showcasing their applicability in estimating time complexity across various applications.

While code embeddings demonstrate significant potential in diverse applications, challenges and limitations exist. Incorporating semantic knowledge into embeddings remains challenging, as models may struggle to interpret contextual or domain-specific information that human programmers easily grasp. Kang et al. [25] investigated the potential benefits of embeddings in downstream tasks for source code models but found no tangible improvement in existing models, calling for further research in this direction.

Romanov and Ivanov [26] experimentally explored the use of pre-trained graph neural networks for type prediction, revealing that pre-training did not enhance type prediction performance. Ding et al. [27] studied the generalizability of pre-trained code embeddings for various software engineering tasks, introducing StrucTexVec, a two-stage unsupervised training framework. Their experiments demonstrated that pre-trained code embeddings, incorporating structural context, could be advantageous in most software engineering tasks.

Another challenge lies in the requirement for substantial amounts of training data. Code embeddings rely on a large corpus of code for training, limiting their applicability in specific programming languages or domains with smaller codebases.

As previously mentioned, NLP transformers, utilizing attention mechanisms, have become dominant in handling sequential data. Unlike traditional models such as recurrent neural networks (RNNs) and long short-term memory (LSTM), transformers, particularly in the form of bidirectional encoder representations from transformers (BERT), have demonstrated superior performance in various NLP tasks, including those related to code.

The self-attention mechanism in transformers can be mathematically described as the following: $\text{Attention(Q,K,V)} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$, where Q represents the query matrix, K represents the key matrix, V represents the value matrix, and d_k represents the dimension of the key vectors.

NLP transformers leverage self-attention mechanisms [28] to process words in a sentence. This involves the model learning to focus on other words in the sentence for each word, assigning weights based on significance. In code-related tasks, the transformer's input comprises a sequence of tokens representing parts of the code (e.g., function names, variable names). These tokens traverse the transformer model, enabling it to learn relationships between different parts of the code. The model then predicts the next token based on the acquired relationships.

A notable advantage of NLP transformers in code-related tasks is their proficiency in handling long sequences [29]. Unlike traditional models like RNNs and LSTMs, transformers circumvent the vanishing gradient problem when processing lengthy sequences, leading to significantly improved performance.

Transformers introduce positional encoding [28], conveying information about the position of words in a sentence. This proves beneficial in code-related tasks where the order of code is crucial for functionality. Positional encoding aids the model in distinguishing between words with similar meanings but from different parts of the code, enhancing overall performance. In Figure 1, we present a timeline of the main contributions of AI-assisted programming tasks.

NLP transformers find successful applications in various code-related tasks. In code completion [30], the model predicts the next tokens of code given a partial snippet. Code

summarization [31] involves generating a concise summary of a piece of code, aiding comprehension of large and complex codebases. Code translation [32] sees the model translating code from one programming language to another, particularly useful for dealing with legacy code.

One prominent NLP transformer model is bidirectional encoder representations from transformers (BERT) [33], widely applied in code-related tasks due to its success in natural language tasks. Another transformer model, gated transformer [34], addresses the limitations of the original transformer, enhancing efficiency on long sequences with repetitive elements.

The transformer architecture consists of encoder and decoder components. The encoder processes input text, converting it into a sequence of vectors, while the decoder generates output text based on these vectors. The encoder comprises multiple identical layers, each featuring self-attention and feed-forward network sub-layers. Input to the encoder passes through an embedding layer, converting the input sequence into fixed-dimensional embeddings. Self-attention within the encoder captures relevant information in the input sequence, utilizing multiple heads or parallel attention mechanisms to attend to different parts of the sequence. These mechanisms compute weighted sums based on word importance, capturing long-term dependencies.

The self-attention and feed-forward network layers repeat within the encoder, allowing hierarchical processing of the input sequence. This hierarchical approach captures different levels of abstraction [33], resulting in the hidden representation of the input sequence for further processing by the decoder.

An advantage of transformers lies in the encoder's ability to process input sequences of variable lengths, offering versatility for various NLP tasks. The use of multiple heads [35] in self-attention mechanisms allows transformers to learn diverse representations of the input sequence, enhancing encoder robustness.

In simple terms, multi-head attention empowers the transformer model to attend to multiple pieces of information simultaneously. Instead of relying on a single attention mechanism, multi-head attention deploys several attention mechanisms in parallel, creating multiple representations of the input sequence. These parallel mechanisms, or "heads", perform the same operation with different sets of parameters, enabling the model to attend to different aspects of the input sequence. Context vectors generated by each head are concatenated, resulting in the final representation of the input sequence.

Pre-trained models in transformers are large neural network architectures pre-trained on extensive text data. These models learn statistical patterns and language structures, allowing them to understand and generate human-like text. Unlike traditional language models, pre-trained transformers use bidirectional attention to consider both previous and future words, providing a better understanding of the overall context.

The effectiveness of transformer models in software engineering tasks relies on domain-specific data availability and the relevance of pre-training data to the target domain. Experimentation and adaptation are crucial for optimal results in diverse software engineering applications [36].

The general process in software engineering tasks using transformers can be outlined in the following steps:

- Data preprocessing: The initial step involves preprocessing the input data, typically through tokenization and vectorization of code snippets. This step is crucial to feed meaningful data into the transformer model.
- Transformer architecture: The transformer model comprises an encoder and a decoder. The encoder processes input data to create a code representation, and the decoder utilizes this representation to generate the code.
- Attention mechanism: Transformers incorporate an attention mechanism, a pivotal element allowing the model to focus on specific parts of the input data while generating the output. This enhances efficiency in handling long sequences and capturing complex dependencies.

- Training the model: Following data preprocessing and setting up the transformer model, the next step involves training the model using backpropagation. Batches of data pass through the model, loss is calculated, and model parameters are updated to minimize the loss.
- Fine-tuning: It is essential to assess its quality and make any necessary adjustments to the model. Fine-tuning may involve retraining on a labeled dataset or adjusting hyperparameters.

CodeBERT [37], a transformer model pre-trained on a comprehensive dataset of source code and natural language, excels in understanding the relationship between code and corresponding comments. It demonstrates state-of-the-art performance in code completion, summarization, and translation tasks, generating accurate and human-like code. CodeBERT follows the underlying architecture of BERT with modifications to suit the programming language domain [38]. The bidirectional transformer encoder takes in code and natural language sequences, encoding them into contextualized representations. A decoder then generates a human-readable description of the code. Code and natural language sequences are concatenated with special tokens to indicate the input type. Pre-trained on extensive data from GitHub, Stack Overflow, Wikipedia, and other sources, CodeBERT undergoes fine-tuning for downstream tasks like code summarization, classification, and retrieval. This transfer learning model adapts to different codebases and programming languages, facilitating code generation and retrieval for non-programmers.

T5 (text-to-text transfer transformer) [39], another large-scale transformer model, caters to various natural language tasks. Pre-trained on diverse datasets, T5 can handle tasks such as translation, summarization, and question answering. It has proven effective in code generation tasks, producing high-quality code with detailed explanations.

GPT-3 (generative pre-trained transformer) [40], developed by OpenAI, excels in natural language generation tasks, including code completion. Its large size and pre-training on a wide range of tasks make it adept at generating code for different programming languages, often matching human writing.

XLNet [41], based on the permutation language model, outperforms BERT in many NLP tasks, including code completion. Similar to BERT, XLNet comprehends code syntax and context well, generating code for various programming languages. CCBERT [42], a deep learning model for generating Stack Overflow question titles, exhibits strong performance in regular and low-resource datasets.

EL-CodeBert [43], a pre-trained model combining programming languages and natural languages, utilizes representational information from each layer of CodeBert for downstream source code-related tasks. Outperforming state-of-the-art baselines in four tasks, EL-CodeBert demonstrates effectiveness in leveraging both programming and natural language information.

Transformers have demonstrated impressive performance across various natural language processing (NLP) tasks and have found successful applications in AI-assisted programming. However, they also exhibit certain inherent weaknesses within this domain. One limitation lies in their capacity for contextual understanding. While transformers excel at capturing context within a fixed-length window, typically around 512 tokens, programming tasks often involve extensive codebases where understanding context beyond this window becomes crucial for accurate analysis and generation. Additionally, transformers lack domain-specific knowledge. Being pre-trained on general-purpose corpora, they may not adequately capture the intricacies and specialized knowledge required for programming tasks. This deficiency can result in suboptimal performance when dealing with programming languages, libraries, and frameworks.

In AI-assisted programming tasks, code embeddings and transformers are closely connected, often complementing each other to enhance the capabilities of programming assistance tools. There are connections between code embeddings and transformers in this context:

- Representation learning: Both code embeddings and transformers aim to learn meaningful representations of code. Code embeddings convert source code into fixed-dimensional vectors, capturing syntactic and semantic information. Similarly, transformers utilize self-attention mechanisms to learn contextual representations of code snippets, allowing them to capture dependencies between different parts of the code.
- Semantic understanding: Code embeddings and transformers facilitate semantic understanding of code. Code embeddings map code snippets into vector representations where similar code fragments are closer in the embedding space, aiding tasks like code search, code similarity analysis, and clone detection. Transformers, with their ability to capture contextual information, excel at understanding the semantics of code by considering the relationships between tokens and their context.
- Feature extraction: Both techniques serve as effective feature extractors for downstream tasks in AI-assisted programming. Code embeddings provide compact representations of code that can be fed into traditional machine learning models or neural networks for tasks like code classification, bug detection, or code summarization. Transformers, on the other hand, extract features directly from code snippets using self-attention mechanisms, enabling end-to-end learning for various programming-related tasks.
- Model architecture: Code embeddings and transformers are often integrated into the same model architecture to leverage their complementary strengths. For instance, models like CodeBERT combine transformer-based architectures with code embeddings to enhance code understanding and generation capabilities. This fusion allows the model to capture both local and global dependencies within code snippets, resulting in more accurate and context-aware predictions.
- Fine-Tuning: Pre-trained transformers, such as BERT or GPT, can be fine-tuned on code-related tasks using code embeddings as input features. This fine-tuning process adapts the transformer's parameters to better understand the specific characteristics of programming languages and code structures, leading to improved performance on programming-related tasks.

In conclusion, the use of code embeddings and transformers in software engineering tasks has witnessed substantial growth. Code embeddings, capturing both syntactic and semantic information, offer effective representation learning techniques. Transformers, particularly in the form of BERT and its derivatives, demonstrate superior performance in various code-related tasks, owing to their ability to handle long sequences and consider both past and future context. The pre-trained models, such as CodeBERT and T5, have shown remarkable success in code generation, summarization, and translation tasks. However, challenges such as incorporating semantic knowledge into embeddings and the need for extensive training data persist. Continuous experimentation and adaptation are crucial for harnessing the full potential of these advanced techniques in diverse software engineering applications.

3. Methodology

A comprehensive review of literature pertaining to AI-supported programming tasks was conducted. The selection criteria were based on both content and publication year. Specifically, papers were chosen based on their utilization of code-embeddings or transformer technologies to facilitate AI-assisted programming tasks. The focus was on papers explicitly mentioning specific programming tasks that were supported. The scope of the research encompassed papers published within the last 5 years.

To ensure a thorough examination, only publications indexed in Scopus were taken into consideration. The identification of relevant papers was achieved through a keyword-based search using terms such as "code embeddings" and "transformers", coupled with specific programming tasks (e.g., "code embeddings bug detection").

4. AI-Supported Programming Tasks

In this section, the current body of literature on AI-assisted programming is examined, emphasizing the specific tasks addressed by the studied approaches. The discussion is organized around a framework comprising nine programming tasks identified in the relevant literature. These tasks encompass code summarization, bug detection and correction, code completion, code generation process, code translation, code comment generation, duplicate code detection and similarity, code refinement, and code security.

4.1. Code Summarization

Code summarization involves generating natural language descriptions for source code written in various programming languages, primarily to support documentation generation. During this process, input source code is transformed into a descriptive narrative, typically in English, providing an overview of the code's functionality at the function level.

An enhanced code embedding approach known as Flow2Vec [16] improved the representation of inter-procedural program dependence (value flows) with precision. It accommodated control flows and data flows with alias recognition, mapping them into a low-dimensional vector space. Experiments on 32 open-source projects demonstrated Flow2Vec's effectiveness in enhancing the performance of existing code embedding techniques for code classification and code summarization tasks.

Transformers play a crucial role in generating summaries, involving preprocessing the text by removing unnecessary characters and segmenting them into smaller sentences or phrases. The transformer model, trained on extensive text data, utilizes its attention mechanism to identify key words and phrases, producing a summary based on these essential elements.

Wang et al. [44] introduced Fret, a functional reinforced transformer with BERT, which outperformed existing approaches in both Java and Python. Achieving a BLEU-4 score of 24.32 and a ROUGE-L score of 40.12, Fret demonstrated superior performance in automatic code summarization. For smart contracts, Yang et al. [45] proposed a multimodal transformer-based code summarization model, showcasing its ability to generate higher-quality code comments compared to state-of-the-art baselines.

Hou et al. [46] presented TreeXFMR, an automatic code summarization paradigm with hierarchical attention, using abstract syntax trees and positional encoding for code representation. Pre-trained and tested on GitHub, TreeXFMR achieved significantly better results than baseline methods.

GypSum [47] incorporated a graph attention network and a pre-trained programming and natural language model for code summarization. Utilizing a dual-copy mechanism, GypSum achieved effective hybrid representations and improved the summary generation process. Gu et al. [48] introduced AdaMo, a method for automated code summarization leveraging adaptive strategies like pre-training and intermediate fine-tuning to optimize latent representations.

Ma et al. [49] proposed a multi-modal fine-grained feature fusion model for code summarization, effectively aligning and fusing information from token and abstract syntax tree modalities. Outperforming current state-of-the-art models, this approach demonstrated superior results.

Gong et al. [31] presented SCRIPT, a structural relative position-guided transformer, using ASTs to capture source code structural dependencies. SCRIPT outperformed existing models on benchmark datasets in terms of BLEU, ROUGE-L, and METEOR metrics. Gao and Lyu [50] proposed M2TS, an AST-based source code summarization technique integrating AST and token features to capture the structure and semantics of source code, demonstrating performance on Java and Python language datasets.

Ferretti and Saletta [51] introduced a novel summarization approach using a pseudo-language to enhance the BRIO model, outperforming CodeBERT and PLBART. The study ex-

plored the limitations of existing NLP-based approaches and suggested further research directions.

Choi et al. [52] presented READSUM, a model combining abstractive and extractive approaches for generating concise and informative code summaries. READSUM considered both structural and temporal aspects of input code, utilizing a multi-head self-attention mechanism to create augmented code representations. The extractive procedure verified the relevancy of important keywords, while the abstractive approach generated high-quality summaries considering both structural and temporal information from the source code.

In summary, code embeddings and transformers both play crucial roles in code summarization, yet they operate in distinct ways. Code embeddings typically involve representing code snippets as fixed-length vectors in a continuous vector space, capturing semantic and syntactic information. This approach offers simplicity and efficiency in handling code representations but may struggle with capturing long-range dependencies. On the other hand, transformers excel in modeling sequential data by processing the entire input sequence simultaneously through self-attention mechanisms. This allows them to capture intricate dependencies across code snippets effectively, resulting in more comprehensive summarizations. However, transformers often require larger computational resources compared to code embeddings. Thus, while code embeddings offer efficiency and simplicity, transformers provide a more powerful and context-aware solution for code summarization tasks.

4.2. Bug Detection and Correction

This task focuses on identifying errors in code (Figure 2), emphasizing the detection of unknown errors to enhance software reliability. Traditional bug detection methods rely on manual code reviews, which are often tedious and time-consuming. In contrast, code embedding presents an efficient approach, capable of processing large volumes of code and identifying potential bugs within minutes. The effectiveness of code embedding depends on a diverse training dataset, as a lack of diversity may hinder its ability to capture all types of bugs.

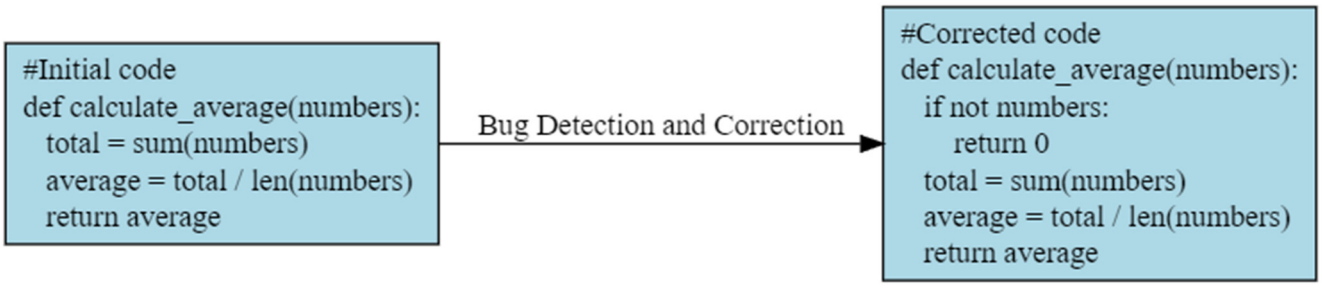

Figure 2. Code bug detection and correction example.

Aladics et al. [53] demonstrated that representing source code as vectors, based on an abstract syntax tree and the Doc2Vec algorithm, improved bug prediction accuracy and was suitable for machine learning tasks involving source code. Cheng et al. [54] proposed a self-supervised contrastive learning approach for static vulnerability detection, leveraging pre-trained path embedding models to reduce the need for labeled data. Their approach outperformed eight baselines for bug detection in real-world projects.

Hegedus and Ferenc [55] used a machine learning model to filter out false positive code analysis warnings from an open-source Java dataset, achieving an accuracy of 91%, an F1-score of 81.3%, and an AUC of 95.3%. NLP transformers offer an efficient and accurate method for bug detection by analyzing source code, identifying patterns, and detecting inconsistencies indicative of bugs. Bagheri and Hegedus [56] compared text representation methods (word2vec, fastText, and BERT) for detecting vulnerabilities in Python code, with BERT exhibiting the highest accuracy rate (93.8%). Gomes et al. [57] found that BERT-based feature extraction significantly outperformed TF-IDF-based extraction in predicting long-

lived bugs, with support vector machines and random forests producing better results when using BERT.

Code summarization, utilizing NLP transformers, presents an approach to bug detection by automatically generating human-readable summaries of code fragments. This method has shown promise in detecting bugs in open-source projects with ample code and bug data available for training.

Evaluation of four new CodeBERT models for predicting software defects demonstrated their ability to improve predictive accuracy across different software versions and projects [58]. The choice of distinct prediction approaches influenced the accuracy of the CodeBERT models.

DistilBERT, a lightweight version of BERT, pre-trained and fine-tuned on various NLP tasks, including bug detection and correction, offers faster and more efficient bug detection, albeit with potentially lower performance than other transformer models. AttSum, a deep attention-based summarization model, surpassed existing models in evaluating bug report titles [59].

Bugsplainer, a transformer-based generative model for explaining software bugs to developers, presented more precise, accurate, concise, and helpful explanations than previous models [60]. Transformers contribute to bug localization, identifying the exact location of bugs in the code. Validation of patches in automated program repair (APR) remains a crucial area, with Csuvik et al. [61] demonstrating the utility of Doc2Vec models in generating patches for JavaScript code.

Mashhadi and Hemmati [62] introduced an automated program repair approach relying on CodeBERT, generating qualitative fixes in various bug cases. Chakraborty et al. [63] created Modit, a multi-modal NMT code editing engine, which outperformed existing models in obtaining correct code patches, especially when developer hints were included.

Generate and validate, a strategy for automatic bug repair using the generative pre-trained transformer (GPT) model, achieved up to 17.25% accuracy [64]. SeqTrans, proposed by Chi et al. [65], demonstrated superior accuracy in addressing certain types of vulnerabilities, outperforming previous strategies in the context of neural machine translation (NMT) technology.

VRepair, an approach by Chen et al. [66], utilized deep learning and transfer learning techniques for automatic software vulnerability repair, showing effectiveness in repairing security vulnerabilities in C. Kim and Yang [67], who utilized the BERT algorithm to predict duplicated bug reports, outperforming existing models and improving bug resolution times.

A technique for developing test oracles, combined with automated testing, improved accuracy by 33%, identifying 57 real-world bugs [68]. da Silva et al. [69] explored various program embeddings and learning models for predictive compilation, with surprisingly simple embeddings performing comparably to more complex ones.

In summary, code embeddings and transformers serve as valuable tools for bug detection and correction, each with its unique strengths. Code embeddings offer a concise representation of code snippets, capturing their semantic and syntactic properties in a fixed-length vector format. This can facilitate efficient similarity comparisons between code segments, aiding in identifying similar bug patterns across projects. However, code embeddings may struggle with capturing complex contextual information and long-range dependencies, potentially leading to limitations in detecting subtle bugs. In contrast, transformers excel in modeling sequential data through self-attention mechanisms, enabling them to capture intricate patterns and contextual information across code segments. This makes transformers particularly effective in detecting and correcting bugs that involve complex interactions and dependencies between code components. Despite the promising results of NLP transformers in bug detection, challenges include the scarcity of large, high-quality datasets and the significant computational resources and training time required. Existing datasets are often language-specific, making generalization to different codebases

challenging. Additionally, the resource-intensive nature of NLP transformers may limit their suitability for real-time bug detection.

4.3. Code Completion

Code completion, a crucial aspect of programming, involves suggesting code to assist programmers in efficiently completing the code they are currently typing. This suggestion can span variable and function names to entire code snippets. The application of transformers in code completion harnesses advanced language models, trained on extensive text data, to enhance developers' coding efficiency. These models exhibit a deep understanding of the context of the code under construction, predicting and suggesting the next code sequence as developers type. This extends beyond basic keyword suggestions, encompassing variable names, function calls, and even the generation of complete code snippets.

The model's proficiency in comprehending syntactic and semantic structures in programming languages ensures accurate and contextually relevant suggestions. It plays a role in identifying and preventing common coding mistakes by offering real-time corrections. Moreover, code completion with transformers often entails providing contextual information such as function signatures, parameter details, and relevant documentation. This not only accelerates the coding process but also aids developers in effectively utilizing various functions and methods.

Roberta [70], another transformer model, has demonstrated impressive results in various natural language processing tasks, showcasing noteworthy performance in code completion. It excels in generating code for diverse programming languages, showcasing a robust understanding of code syntax and context.

Transformer-XL [71], designed to handle longer sequences compared to traditional transformers, has exhibited promising outcomes in code completion tasks, especially when dealing with extensive and intricate sequences. It showcases proficiency in generating code for various programming languages.

CodeFill, proposed by Izadi et al. [72], is a language model for autocompletion leveraging learned structure and naming information. Outperforming several baseline and state-of-the-art models, including GPT-C and TravTrans+, CodeFill excels in both single-token and multi-token prediction. All code and datasets associated with CodeFill are publicly available.

CCMC, presented by Yang and Kuang [29], is a code completion model utilizing a Transformer-XL model for handling long-range dependencies and a pointer network with CopyMask for copying OOV tokens from inputs. The model demonstrates excellent performance in code completion on real-world datasets.

Developers can seamlessly integrate code completion into their preferred integrated development environments (IDEs) or code editors, enhancing the overall coding experience. The interactive and adaptive nature of transformer-based code completion renders it a powerful tool for developers working across various programming languages and frameworks.

Liu et al. [73] introduced a multi-task learning-based pre-trained language model with a transformer-based neural architecture to address challenges in code completion within integrated development environments (IDEs). Experimental results highlight the effectiveness of this approach compared to existing state-of-the-art methods.

BART (bidirectional and auto-regressive transformer), another popular transformer model developed [74], is trained using a combination of supervised and unsupervised learning techniques. Specifically designed for text generation tasks, BART has shown promising results in code generation, achieving state-of-the-art performance in code completion tasks where it predicts the remaining code based on the given context.

A novel neural architecture based on transformer models was proposed and evaluated for autocomplete systems in IDEs, showcasing an accuracy increase of 14–18%. Additionally, an open-source code and data pipeline were released [75]. While transformer models exhibit promise for code completion, further enhancements in accuracy are essential for addressing complex scenarios [30].

In summary, code embeddings and transformers are both valuable tools for code completion, each offering distinct advantages. Code embeddings provide a compact representation of code snippets in a continuous vector space, capturing their semantic and syntactic properties. This allows for efficient retrieval of similar code segments, aiding in suggesting relevant completions based on the context of the code being written. However, code embeddings may struggle with capturing long-range dependencies and contextual nuances, potentially leading to less accurate suggestions in complex coding scenarios. Transformers, on the other hand, excel in modeling sequential data through self-attention mechanisms, enabling them to capture intricate patterns and contextual information across code sequences. This results in more accurate and context-aware code completions, especially in scenarios where understanding broader context and dependencies is crucial.

4.4. Code Generation Process

Code generation involves the task of creating source code based on constraints specified by the programmer in natural language. Hu et al. [23] introduced a supervised code embedding approach along with a tree representation of code snippets, demonstrating enhanced accuracy and efficiency in generating code from natural language compared to current state-of-the-art methods.

Transformers, a type of neural network architecture widely used for various natural language processing (NLP) tasks, including code generation, utilize an attention mechanism to capture long-term dependencies. They excel in handling sequential data without relying on recurrent connections, making them well-suited for tasks involving code generation.

Transformers can be applied to generate functions or methods based on high-level specifications. Developers can articulate the desired functionality in natural language, and the transformer generates the corresponding code.

Svyatkovskiy et al. [3] introduced IntelliCode Compose, a versatile, multilingual code completion tool capable of predicting arbitrary code tokens and generating correctly structured code lines. It was trained on 1.2 billion lines of code across four languages and utilized in the Visual Studio Code IDE and Azure Notebook.

Gemmell et al. [76] explored Transformer architectures for code generation beyond existing IDE capabilities, proposing a "Relevance Transformer" model. Benchmarking results demonstrated improvement over the current state-of-the-art.

Soliman et al. [77] presented MarianCG-NL-to-Code, a code generation transformer model for generating Python code from natural language descriptions. Outperforming state-of-the-art models, it was downloadable on GitHub and evaluated on CoNaLa and DJANGO datasets.

ExploitedGen [78], an exploit code generation approach based on CodeBERT, achieved better accuracy in generating exploit code than existing methods. It incorporated a template-augmented parser and a semantic attention layer, with additional experiments assessing generated code for syntax and semantic accuracy.

Laskari et al. [79] discussed Seq2Code, a transformer-based solution for translating natural language problem statements into Python source code. Using an encoder–decoder transformer design with multi-head attention and separate embeddings for special characters, the model demonstrated improved perplexity compared to similarly structured models.

To summarize the code generation process, code embeddings and transformers offer distinctive approaches, each with its own strengths. Code embeddings condense code snippets into fixed-length vectors, capturing semantic and syntactic information efficiently. This simplifies the generation process by enabling quick retrieval of similar code segments and facilitating straightforward manipulation in vector space. However, code embeddings might struggle with capturing complex dependencies and contextual nuances, potentially limiting their ability to produce diverse and contextually accurate code. In contrast, transformers excel in modeling sequential data through self-attention mechanisms, allowing them to capture intricate patterns and long-range dependencies across code sequences.

This enables transformers to generate code with greater context awareness and flexibility, resulting in more accurate and diverse outputs. Nevertheless, transformers typically demand significant computational resources and extensive training data compared to code embeddings.

4.5. Code Translation

Code translation (Figure 3) involves the conversion of source code from one programming language to another, commonly employed for managing legacy source code. Unlike code generation, which takes natural language as input, code translation deals directly with source code. Bui et al. [80] introduced a bilingual neural network (Bi-NN) architecture for automatically classifying Java and C++ programs. Comprising two sub-networks dedicated to Java and C++ source code, Bi-NN utilized an additional neural network layer to recognize similarities in algorithms and data structures across different languages. Evaluation of a code corpus containing 50 diverse algorithms and data structures revealed promising classification results, with increased accuracy attributed to encoding more semantic information from the source code.

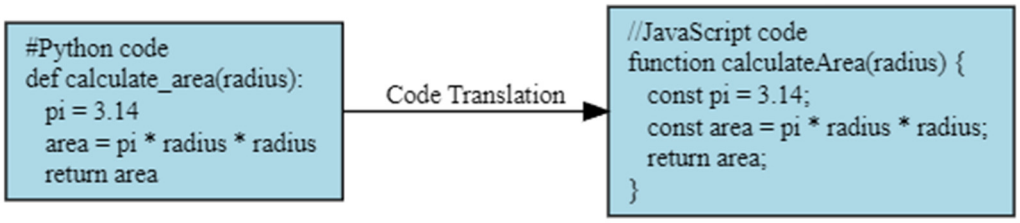

Figure 3. Code translation example.

In contrast to traditional machine translation methods, transformers, which employ self-attention mechanisms instead of recurrent networks, play a pivotal role in code translation. Transformers facilitate the automatic conversion of source code written in one programming language into its equivalent in another language. This capability proves valuable for tasks such as cross-language code migration, integrating code from different languages, or aiding developers familiar with one language in comprehending and working with code written in another.

Hassan et al. [32] introduced a source code converter based on the neural machine translation transformer model, specializing in converting source code between Java and Swift. The model was trained on a merged dataset, and initial results demonstrated promise in terms of the pipeline and code synthesis procedure.

DeepPseudo, presented by Yang et al. [81], leveraged advancements in sequence-to-sequence learning and code semantic learning to automatically generate pseudo-code from source code. Experiment results indicated DeepPseudo's superiority over seven state-of-the-art models, providing a valuable tool for novice developers to understand programming code more easily.

Alokla et al. [82] proposed a new model for generating pseudocode from source code, achieving higher accuracy compared to previous models. This model utilized similarity measures and deep learning transformer models, demonstrating promising results on two datasets.

DLBT, a deep learning-based transformer model for automatically generating pseudocode from source code [83], tokenized the source code and employed a transformer to assess the relatedness between the source code and its corresponding pseudocode. Tested with Python source code, DLBT achieved accuracy and BLEU scores of 47.32 and 68.49, respectively.

Acharjee et al. [84] suggested a method utilizing natural language processing and a sequence-to-sequence deep learning-based model trained on the SPoC dataset for pseudocode conversion. This approach exhibited increased accuracy and efficiency compared to other techniques, as evaluated using bilingual understudy scoring.

To sum up regarding the realm of code generation translation, both code embeddings and transformers offer distinct advantages. Code embeddings condense code snippets into fixed-length vectors, effectively capturing the semantic and syntactic information essential for translation tasks. This approach simplifies the translation process by enabling quick retrieval of similar code segments and facilitating straightforward manipulation in vector space. However, code embeddings may struggle to capture complex dependencies and nuances present in code, potentially limiting their ability to produce accurate translations. On the other hand, transformers excel in modeling sequential data through self-attention mechanisms, allowing them to capture intricate patterns and long-range dependencies across code sequences. This results in more context-aware translations, with the ability to handle a wide range of coding languages and structures.

4.6. Code Comment Generation

The objective of this task is the automatic generation of natural language comments for a given code snippet. Shahbazi et al. [85] introduced API2Com, a comment generation model that utilized Application Programming Interface Documentations (API Docs) as external knowledge resources. The authors observed that API Docs could enhance comment generation, especially when there was only one API in the method. However, as the number of APIs increased, the model output was negatively impacted.

ComFormer, proposed by Yang et al. [86], is a novel code comment generator that integrates transformer and fusion method-based hybrid code presentation. Byte-BPE and Sim_SBT were employed to address out-of-vocabulary (OOV) problems during training. The evaluation involved three metrics and a human study comparing ComFormer to seven state-of-the-art baselines from both code comment and neural machine translation (NMT) domains.

Chakraborty et al. [87] introduced a new pre-training objective for language models for source code, aiming to naturalize the code by utilizing its bi-channel structure (formal and informal). The authors employed six categories of semantic maintaining changes to construct unnatural forms of code for model training. After fine-tuning, the model performed on par with CodeT5, exhibiting improved performance for zero-shot and few-shot learning, as well as better comprehension of code features.

Geng et al. [88] proposed a two-stage method for creating natural language comment texts for code. The approach utilized a model interpretation strategy to refine summaries, enhancing accuracy. Thongtanunam et al. [89] developed AutoTransform, an advanced neural machine translation (NMT) model that significantly increased accuracy in automatically transforming code for code review processes. This innovation aimed to reduce developers' time and effort in manual code review.

BASHEXPLAINER [90] automated code comment generation for Bash scripts, outperforming existing methods based on metrics such as BLEU-3/4, METEOR, and ROUGE-L by up to 9.29%, 8.75%, 4.77%, and 3.86%, respectively. Additionally, it offered a browser plug-in to facilitate the understanding of Bash code.

S-Coach, presented by Lin et al. [91], is a two-phase approach to updating software comments. The first phase utilizes a predictive model to determine if comment updates are code-indicative. If affirmative, an off-the-shelf heuristic-based approach is employed; otherwise, a specially-designed deep learning model is leveraged. Results demonstrated that this approach is more effective than the current state-of-the-art by 20%.

In the domain of code comment generation, both code embeddings and transformers play vital roles, each offering distinct advantages. Code embeddings provide a concise representation of code snippets in a continuous vector space, capturing their semantic and syntactic properties. This facilitates the generation of comments by enabling efficient retrieval of similar code segments and assisting in understanding the context for comment generation. However, code embeddings may struggle with capturing the intricacies and nuances of code, potentially leading to less contextually relevant comments. Transformers, on the other hand, excel in modeling sequential data through self-attention mechanisms,

allowing them to capture complex patterns and dependencies across code sequences. This results in more context-aware and informative comments that better align with the underlying code logic.

4.7. Duplicate Code Detection and Similarity

This task involves identifying duplicate code snippets, whether within the same codebase or across different codebases. Transformers play a crucial role in duplicate code detection, automating the identification of redundant or duplicated code segments within a software project. This process is vital for maintaining code quality, enhancing maintainability, and preventing potential issues associated with code redundancy.

Karakatic et al. [92] introduced a novel method for comparing software systems by computing the robust Hausdorff distance between semantic source code embeddings of each program component. The authors utilized a pre-trained neural network model, code2vec, to generate source code vector representations from various open-source libraries. Employing different types of robust Hausdorff distance, the proposed method demonstrated its suitability for gauging semantic similarity.

The presence of code smells and security smells in various training datasets, a fine-tuned transformer-based GPT-Neo model, and a closed-source code generation tool raised concerns about the cautious application of language models to code generation tasks [93].

Yu et al. [94] proposed BEDetector, a two-channel feature extraction method for binary similarity detection, encompassing contextual semantic feature extraction and a neural GAE model. This system achieved impressive detection rates, including 88.8%, 86.7%, and 100% for resilience against CVE vulnerabilities ssl3-get-key-exchange, ssl3-get-new-session-ticket, and udhcp-get-option, respectively.

Mateless et al. [95] developed Pkg2Vec to encode software packages and predict their authors with remarkable accuracy. Comparisons against state-of-the-art algorithms on the ISOT datasets revealed Pkg2Vec's superior performance, showcasing a 13% increase in accuracy. This demonstrated the efficacy of applying deep learning to improve authorship attribution of software packages, providing deep, interpretable features indicating the unique style and intentions of the programmer.

CodeBERT showed effectiveness for Type-1 and Type-4 clone detection, although its performance declined for unseen functionalities. Fine-tuning was identified as a potential avenue to marginally improve recall [96]. Kovacevic et al. [97] investigated the effectiveness of both ML-based and heuristics-based code smell detection models, utilizing different source code representations (metrics and code embeddings) on the large-scale MLCQ dataset. Transfer learning models were evaluated to analyze the impact of mined knowledge on code smell detection.

An efficient transformer-based code clone detection method was proposed by [98], promising accurate and rapid identification of code clones while significantly reducing computational cost.

To sum up, in the realm of duplicate code detection and similarity analysis, both code embeddings and transformers offer unique advantages. Code embeddings distill code snippets into fixed-length vectors, effectively capturing their semantic and syntactic features. This enables efficient comparison and retrieval of similar code segments, facilitating the identification of duplicate code instances. However, code embeddings may struggle to capture complex dependencies and contextual nuances, potentially limiting their effectiveness in detecting subtle similarities. Transformers, on the other hand, excel in modeling sequential data through self-attention mechanisms, allowing them to capture intricate patterns and long-range dependencies across code sequences. This results in more accurate and context-aware similarity analysis, enabling the detection of subtle variations and similarities within code snippets. Nonetheless, transformers typically require larger computational resources and extensive training data compared to code embeddings.

4.8. Code Refinement

Code refinement (Figure 4) involves identifying and correcting pieces of code susceptible to bugs or vulnerabilities. In the work of Liu et al. [99], a software maintenance method was introduced for debugging method names by evaluating the consistency between their names and code to identify discrepancies. Through experiments on over 2.1 million Java methods, the method achieved an F1-measure of 67.9%, surpassing existing techniques by 15%. Notably, the authors successfully fixed 66 inconsistent method names in a live study on projects in the wild.

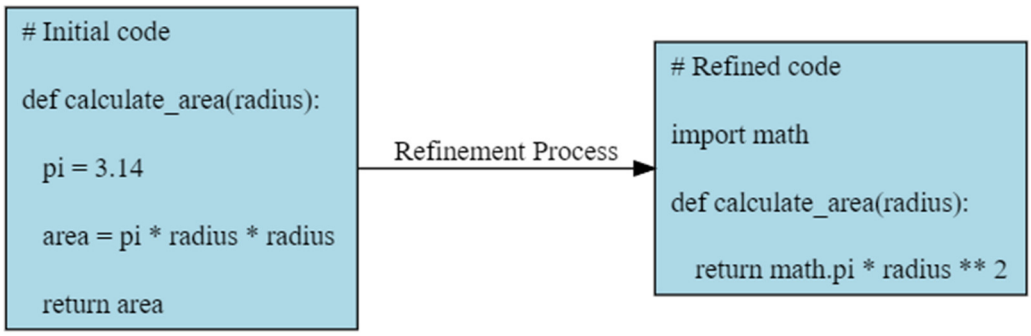

Figure 4. Code refinement example.

Cabrera Lozoya et al. [100] extended a state-of-the-art approach for representing source code to also include changes in the source code (commits). Transfer learning was then applied to classify security-relevant commits. The study demonstrated that representations based on structural information of the code syntax outperformed token-based representations. Moreover, pre-training with a small dataset (greater than 10^4 samples) for a closely related pretext task showed superior performance compared to pre-training with a larger dataset (more than 10^6 samples) and a loosely related pretext task.

Wang et al. [101] introduced Cognac, a context-guidance method name recommender that incorporated global context from methods related by calls. It utilized prior knowledge to adjust method name recommendations and method name consistency checking tasks. Cognac outperformed existing approaches on four datasets with F-scores of 63.2%, 60.8%, 66.3%, and 68.5%, respectively, achieving an overall accuracy of 76.6%, surpassing MNire by 11.2%, a machine learning approach to check the consistency between the name of a given method and its implementation [102].

Xie et al. [103] proposed DeepLink, a model applying code knowledge graph embeddings and deep learning to identify links between issue reports and code commits for software projects. Evaluation of real-world projects demonstrated its superiority over current state-of-the-art solutions.

Borovits et al. [104] presented an automated procedure using word embeddings and deep learning processes to detect inconsistencies between infrastructure as code (IaC) code units and their names. Experiments on an open-source dataset showed an accuracy range of 78.5% to 91.5% in finding such inconsistencies.

Ma et al. [105] introduced Graph-code2vec, a novel self-supervised pre-training approach using code investigation and graph neural networks to generate agnostic task embeddings for software engineering tasks. The proposed technique proved more effective than existing generic and task-specific learning-based baselines, including GraphCodeBERT.

NaturalCC [106] is an open-source code intelligence toolkit, accessible on the website (http://xcodemind.github.io), built on Fairseq and PyTorch technology. It is designed to enable efficient machine learning-based implementation of code intelligence tasks such as code summarization, code retrieval, and code completion.

In the context of code refinement, both code embeddings and transformers offer distinct advantages. Code embeddings condense code snippets into fixed-length vectors, capturing their semantic and syntactic properties efficiently. This facilitates the refinement

process by enabling quick retrieval of similar code segments and aiding in identifying areas for improvement. However, code embeddings may struggle to capture complex dependencies and nuanced coding patterns, potentially limiting their ability to suggest refined solutions accurately. Conversely, transformers excel in modeling sequential data through self-attention mechanisms, enabling them to capture intricate patterns and dependencies across code sequences. This results in more contextually aware refinements, with the ability to suggest solutions that align closely with the underlying logic of the code.

4.9. Code Security

Code security involves checking source code for exploits that may allow unauthorized access to restricted resources. Zaharia et al. [107] proposed the use of an intermediate representation that strikes a balance between stringency to retain security flaws, as per MITRE standards, and dynamism that does not strictly rely on the lexicon of a programming language. This intermediate representation is based on the semantical clusterization of commands in C/C++ programs through word embeddings. These embeddings are distributed through the formed intermediate representation to different classifiers for recognizing security vulnerability patterns.

In related work, Zaharia et al. [108] developed a security scanning system employing machine learning algorithms to detect various patterns of vulnerabilities listed in the Common Weaknesses Enumeration (CWE) from NIST. This system, independent of the programming language, achieved a recall value exceeding 0.94, providing a robust defense against cyber-attacks.

Barr et al. [109] conducted an in-depth analysis of the Fluoride Bluetooth module's source code using deep learning, machine learning, heuristics, and combinatorial optimization techniques. They employed byte-pair encoding to lower dimensionality, embedded tokens into a low-dimensional Euclidean space using LSTM, and created a distance matrix based on cosines between vectors of functions. The authors used cluster-editing to segment the graph's vertices into nearly complete subgraphs, assessing vulnerability risk based on vectors and features of each component.

Saletta and Ferretti [110] discussed a technique using natural language processing to recognize security weaknesses in source code. This involved mapping code to vector space through its abstract syntax trees, and supervised learning to capture distinguishing features among different vulnerabilities. Results demonstrated the model's ability to accurately recognize various types of security weaknesses.

In the domain of code security, both code embeddings and transformers serve as valuable tools, each with its unique strengths. Code embeddings offer a compact representation of code snippets, capturing their semantic and syntactic properties efficiently. This allows for quick analysis of code similarities, aiding in the identification of potential security vulnerabilities based on patterns observed in known security issues. However, code embeddings may struggle to capture complex interactions and subtle security flaws, potentially leading to limitations in detecting sophisticated attacks. Transformers, on the other hand, excel in modeling sequential data and understanding contextual information through self-attention mechanisms. This enables them to capture intricate patterns and dependencies across code sequences, resulting in a more comprehensive and context-aware analysis of code security. However, transformers typically require larger computational resources and extensive training data compared to code embeddings.

5. Datasets

Similar to most deep learning models, transformers demand extensive data to exhibit optimal performance. This becomes a notable challenge in the field of programming, where acquiring high-quality datasets is not as straightforward as in natural language processing (NLP).

To tackle this issue, initiatives like CodeSearchNet (https://github.com/github/CodeSearchNet, accessed on 10 January 2024) and CodeXGLUE (https://github.com/

microsoft/CodeXGLUE, accessed on 10 January 2024) have been established, providing valuable datasets for training and evaluating code-related models. CodeSearchNet stands out as a large-scale dataset, encompassing over 6 million GitHub repositories and 4.2 million code files. It spans six programming languages: Java, Python, JavaScript, Go, Ruby, and PHP. CodeBERT has undergone training on this comprehensive dataset, enhancing its capacity to learn cross-lingual representations of source code.

CodeXGLUE, on the other hand, serves as a benchmark dataset strategically crafted for the advancement and assessment of code intelligence methods, specifically focusing on code completion and code retrieval tasks. This dataset incorporates 14 tasks across various programming languages such as Python, Java, C++, and PHP. CodeBERT, recognizing the significance of diverse challenges, has undergone training on this dataset to elevate its proficiency in code intelligence tasks.

6. Conclusions

Code embeddings serve as vector representations of source code, acquired through deep learning techniques. They adeptly encapsulate the lexical, syntactic, and semantic intricacies of code, projecting them into a high-dimensional vector space. Various methods, including recurrent neural networks (RNNs), convolutional neural networks (CNNs), and graph neural networks (GNNs), are employed to generate code embeddings. These methods utilize input source code to establish a mapping between code tokens and their corresponding vector representations. Subsequently, the vector representations become inputs for downstream natural language processing (NLP) tasks.

Code embeddings prove formidable in capturing the semantic essence of code, distinguishing themselves from traditional approaches reliant on handcrafted features. Unlike their predecessors, code embeddings autonomously learn semantic relationships between distinct code tokens, enhancing efficiency in grasping nuances like variable and function dependencies. Furthermore, code embeddings exhibit language agnosticism, enabling training on diverse programming languages and proving valuable for tasks demanding code comprehension across language boundaries. Their capacity to generalize effectively to unseen code snippets stems from training on extensive code corpora, enabling the absorption of general patterns and structures prevalent in code.

Transformers, distinguished by their self-attention mechanisms, have excelled in learning from substantial datasets in an end-to-end manner, eliminating the need for task-specific feature engineering. This adaptability allows a single transformer model to assist in multiple programming tasks, facilitated by fine-tuning specific languages or tasks. Nevertheless, the performance of a transformer model fine-tuned for one task may not seamlessly translate to another task without further adaptation.

The application of transformers extends from natural language processing (NLP) to code representation and generation tasks. Self-attention mechanisms empower transformers to discern long-range dependencies within input text, enhancing their ability to capture contextual nuances in code.

In Table 1, we summarize the literature review presented in this paper.

Next, we try to compare the use of code embeddings and transformers in the nine referred tasks.

- Code summarization:
  - Code embeddings capture the semantic meaning of code snippets, enabling summarization through techniques like clustering or similarity-based retrieval.
  - Transformers can learn contextual representations of code, allowing them to generate summaries by attending to relevant parts of the code and its surrounding context.
- Bug detection and correction:
  - By learning embeddings from code, similarity metrics can be applied to detect similar code segments containing known bugs, or to identify anomalous patterns.

    - Transformers can learn to detect bugs by learning from labeled data, and they can also be fine-tuned for specific bug detection tasks. For bug correction, they can generate patches by learning from examples of fixed code.
- Code completion:
    - Embeddings can be used to predict the next tokens in code, enabling code completion by suggesting relevant completions based on learned representations.
    - Transformers excel at predicting sequences and can provide context-aware code completions by considering the surrounding code.
- Code generation:
    - Code embeddings can be used to generate code by sampling from the learned embedding space, potentially leading to diverse outputs.
    - Transformers can generate code by conditioning on input sequences and generating output sequences token by token, allowing for precise control over the generation process.
- Code translation:
    - Embeddings can be leveraged for mapping code from one programming language to another by aligning representations of similar functionality across languages.
    - Transformers can be trained for sequence-to-sequence translation tasks, allowing for direct translation of code between different programming languages.
- Code comment generation:
    - By learning embeddings from code-comment pairs, embeddings can be used to generate comments for code by predicting the most likely comment given the code.
    - Transformers can be trained to generate comments by conditioning on code and generating natural language descriptions, capturing the context and intent of the code.
- Duplicate code detection and similarity:
    - Similarity metrics based on embeddings can efficiently identify duplicate or similar code snippets by measuring the distance between their embeddings.
    - Transformers can learn contextual representations of code, enabling them to identify duplicate or similar code snippets by comparing their representations directly.
- Code refinement:
    - Embeddings can be used to refine code by suggesting improvements based on learned representations and similarity to high-quality code.
    - Transformers can be fine-tuned for code refinement tasks, such as code formatting or refactoring, by learning from labeled data or reinforcement learning.
- Code security:
    - Embeddings can be utilized for detecting security vulnerabilities by identifying patterns indicative of vulnerabilities or by comparing code snippets to known vulnerable code.
    - Transformers can be trained to detect security vulnerabilities by learning from labeled data, and they can also be used for code analysis to identify potential security risks through contextual understanding.

Finally, for AI-assisted programming tasks, leveraging both code embeddings and transformers can significantly enhance the efficiency and effectiveness of the development process. By combining the strengths of both techniques, developers can benefit from a comprehensive AI-assisted programming environment that offers efficient code analysis, accurate recommendations, and context-aware assistance throughout the development lifecycle. This hybrid approach ensures that developers can leverage the simplicity and efficiency of code embeddings alongside the contextual awareness and sophistication of transformers, thereby maximizing productivity and code quality.

Table 1. Literature overview.

Tasks	Publications
Code summarization	[16,43–45,48–51]—Code embedding [31,46,47,52]—Transformer
Bug detection and correction	[53–57,61,68,69]—Code embedding [38,58–60,62–67]—Transformer
Code completion	[29,30,71–75]—Transformer
Code generation process	[23]—Code embedding [3,76–79]—Transformer
Code translation	[80,81,84]—Code embedding [32,82,83]—Transformer
Code comment generation	[85,87,88,90]—Code embedding [86]-Code embedding—Transformer [37,89]—Transformer [91]—Custom
Duplicate code detection and similarity	[92,94,95]—Code embedding [92,96,98]—Transformer [97]—Custom
Code refinement	[99–105]—Code embedding [106]—Transformer
Code security	[107–110]—Code embedding

Ethical Considerations

Various ethical considerations come to the forefront when employing transformers, or any form of AI, for programming tasks. These considerations encompass aspects related to privacy, bias, transparency, and accountability [111].

A primary ethical concern centers around the potential invasion of privacy inherent in the utilization of transformers for programming. Given that transformers are engineered to analyze and process extensive datasets, including personal or sensitive information, questions arise concerning the storage, utilization, and safeguarding of these data. A critical aspect involves ensuring individuals are informed about the use of their information for programming purposes.

Another ethical dimension revolves around the prospect of bias within the data used for training the transformer. Should the analyzed data exhibit biases or gaps, it could profoundly impact the decisions made by the transformer, potentially perpetuating existing biases and fostering discrimination. Therefore, it becomes imperative to curate training data that are diverse, representative, and devoid of bias.

Transparency emerges as a pivotal ethical consideration in the integration of transformers for AI-assisted programming tasks. Programmers must possess a comprehensive understanding of the inner workings of the transformer and the rationale behind its decisions. Transparency serves not only debugging and troubleshooting purposes but also acts as a safeguard against the occurrence of unethical or harmful decisions.

Moreover, accountability assumes a critical role in the ethical framework surrounding the use of transformers in programming. With advancing technology, ascertaining responsibility for the decisions made by a transformer becomes increasingly challenging. In scenarios involving errors or ethical breaches, establishing clear frameworks for accountability and liability becomes indispensable. These frameworks serve to assign responsibility and address any ensuing issues with precision and fairness.

Author Contributions: Conceptualization, S.K.; methodology, S.K.; investigation, S.K., M.T. and V.V.; resources, S.K.; data curation, M.T.; writing—original draft preparation, S.K.; writing—review and editing, M.T.; supervision, V.V.; project administration, V.V. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Hindle, A.; Barr, E.T.; Su, Z.; Gabel, M.; Devanbu, P. On The Naturalness of Software. In Proceedings of the 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 837–847.
2. Shani, I. Survey Reveals AI's Impact on the Developer Experience. 2023. Available online: https://github.blog/2023-06-13-survey-reveals-ais-impact-on-the-developer-experience (accessed on 24 December 2023).
3. Svyatkovskiy, A.; Deng, S.K.; Fu, S.; Sundaresan, N. IntelliCode compose: Code generation using transformer. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Online, 8–13 November 2020. [CrossRef]
4. Bird, C.; Ford, D.; Zimmermann, T.; Forsgren, N.; Kalliamvakou, E.; Lowdermilk, T.; Gazit, I. Taking Flight with Copilot. *Commun. ACM* **2023**, *66*, 56–62. [CrossRef]
5. Friedman, N. Introducing GitHub Copilot: Your AI Pair Programmer. 2021. Available online: https://github.com/features/copilot (accessed on 24 December 2023).
6. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H.P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating large language models trained on code. *arXiv* **2021**, arXiv:2107.03374. [CrossRef]
7. Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. Competition-level Code Generation with Alphacode. *Science* **2022**, *378*, 1092–1097. [CrossRef] [PubMed]
8. Parashar, B.; Kaur, I.; Sharma, A.; Singh, P.; Mishra, D. Revolutionary transformations in twentieth century: Making AI-assisted software development. In *Computational Intelligence in Software Modeling*; De Gruyter: Berlin, Germany, 2022. [CrossRef]
9. Gulwani, S. AI-assisted programming: Applications, user experiences, and neuro-symbolic techniques (keynote). In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 14–18 November 2022. [CrossRef]
10. Vaithilingam, P.; Zhang, T.; Glassman, E.L. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In Proceedings of the CHI Conference on Human Factors in Computing Systems Extended Abstracts, New Orleans, LA, USA, 29 April–5 May 2022; Association for Computing Machinery: New York, NY, USA, 2022; pp. 1–7.
11. Fernandez, R.C.; Elmore, A.J.; Franklin, M.J.; Krishnan, S.; Tan, C. How Large Language Models Will Disrupt Data Management. *Proc. VLDB Endow.* **2023**, *16*, 3302–3309. [CrossRef]
12. Zhou, H.; Li, J. A Case Study on Scaffolding Exploratory Data Analysis for AI Pair Programmers. In Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, Hamburg, Germany, 23–28 April 2023; pp. 1–7. [CrossRef]
13. Kazemitabaar, M.; Chow, J.; Ma, C.K.T.; Ericson, B.J.; Weintrop, D.; Grossman, T. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, Hamburg, Germany, 23–28 April 2023; pp. 1–23. [CrossRef]
14. Daun, M.; Brings, J. How ChatGPT Will Change Software Engineering Education. In Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, Turku, Finland, 7–12 July 2023; pp. 110–116. [CrossRef]
15. Prather, J.; Reeves, B.N.; Denny, P.; Becker, B.A.; Leinonen, J.; Luxton-Reilly, A.; Powell, G.; Finnie-Ansley, J.; Santos, E.A. "It's Weird That It Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *ACM Trans. Comput. Interact.* **2023**, *31*, 1–31. [CrossRef]
16. Sui, Y.; Cheng, X.; Zhang, G.; Wang, H. Flow2Vec: Value-flow-based precise code embedding. *Proc. ACM Program. Lang.* **2020**, *4*, 233. [CrossRef]
17. Rabin, M.R.I.; Mukherjee, A.; Gnawali, O.; Alipour, M.A. Towards demystifying dimensions of source code embeddings. In Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages, Online, 8–13 November 2020. [CrossRef]
18. Azcona, D.; Arora, P.; Hsiao, I.-H.; Smeaton, A. user2code2vec: Embedding for Profiling Students Based on Distributinal Representations of Source Code. In Proceedings of the 9th International Conference on Learning Analytics and Knowledge, Tempe, AZ, USA, 4–8 March 2019. [CrossRef]
19. Ding, Z.; Li, H.; Shang, W.; Chen, T.-H. Towards Learning Generalizable Code Embeddings Using Task-agnostic Graph Convolutional Networks. *ACM Trans. Softw. Eng. Methodol.* **2023**, *32*, 48. [CrossRef]
20. Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; et al. Transformers: State-of-the-Art Natural Language Processing. In *EMNLP 2020—Conference on Empirical Methods in Natural Language Processing: Systems Demonstrations*; Association for Computational Linguistics: Kerrville, TX, USA, 2020; pp. 38–45.

21. Chirkova, N.; Troshin, S. Empirical study of transformers for source code. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, 23–28 August 2021. [CrossRef]
22. Song, Y.; Shi, S.; Li, J.; Zhang, H. Directional skip-gram: Explicitly distinguishing left and right context forword embeddings. In Proceedings of the NAACL HLT 2018—2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, New Orleans, LA, USA, 1–6 June 2018; pp. 175–180.
23. Hu, H.; Chen, Q.; Liu, Z. Code Generation from Supervised Code Embeddings. In *Neural Information Processing*; Springer: Cham, Switzerland, 2019; pp. 388–396. [CrossRef]
24. Sikka, J.; Satya, K.; Kumar, Y.; Uppal, S.; Shah, R.R.; Zimmermann, R. Learning Based Methods for Code Runtime Complexity Prediction. In *Advances in Information Retrieval*; Springer: Cham, Switzerland, 2020; pp. 313–325. [CrossRef]
25. Kang, H.J.; Bissyande, T.F.; Lo, D. Assessing the Generalizability of Code2vec Token Embeddings. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019. [CrossRef]
26. Romanov, V.; Ivanov, V. Prediction of Types in Python with Pre-trained Graph Neural Networks. In Proceedings of the 2022 Ivannikov Memorial Workshop (IVMEM), Moscow, Russia, 23–24 September 2022. [CrossRef]
27. Ding, Z.; Li, H.; Shang, W.; Chen, T.-H.P. Can pre-trained code embeddings improve model performance? Revisiting the use of code embeddings in software engineering tasks. *Empir. Softw. Eng.* **2022**, *27*, 63. [CrossRef]
28. Shaw, P.; Uszkoreit, J.; Vaswani, A. Self-attention with relative position representations. In Proceedings of the NAACL HLT 2018—2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, New Orleans, LA, USA, 1–6 June 2018; pp. 464–468.
29. Yang, H.; Kuang, L. CCMC: Code Completion with a Memory Mechanism and a Copy Mechanism. In Proceedings of the EASE 2021: Evaluation and Assessment in Software Engineering, Trondheim, Norway, 21–23 June 2021. [CrossRef]
30. Ciniselli, M.; Cooper, N.; Pascarella, L.; Mastropaolo, A.; Aghajani, E.; Poshyvanyk, D.; Di Penta, M.; Bavota, G. An Empirical Study on the Usage of Transformer Models for Code Completion. *IEEE Trans. Softw. Eng.* **2021**, *48*, 4818–4837. [CrossRef]
31. Gong, Z.; Gao, C.; Wang, Y.; Gu, W.; Peng, Y.; Xu, Z. Source Code Summarization with Structural Relative Position Guided Transformer. In Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 15–18 March 2022. [CrossRef]
32. Hassan, M.H.; Mahmoud, O.A.; Mohammed, O.I.; Baraka, A.Y.; Mahmoud, A.T.; Yousef, A.H. Neural Machine Based Mobile Applications Code Translation. In Proceedings of the 2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES), Giza, Egypt, 24–26 October 2020. [CrossRef]
33. Devlin, J.; Chang, M.-W.; Lee, K.; Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of the NAACL HLT 2019—2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Minneapolis, MN, USA, 2–7 June 2019; pp. 4171–4186.
34. Sengupta, A.; Kumar, A.; Bhattacharjee, S.K.; Roy, S. Gated Transformer for Robust De-noised Sequence-to-Sequence Modelling. In Proceedings of the 2021 Findings of the Association for Computational Linguistics, Punta Cana, Dominican Republic, 7–11 November 2021.
35. Wu, C.; Wu, F.; Ge, S.; Qi, T.; Huang, Y.; Xie, X. Neural news recommendation with multi-head self-attention. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and 9th International Joint Conference on Natural Language Processing, Hong Kong, China, 3–7 November 2019.
36. Chernyavskiy, A.; Ilvovsky, D.; Nakov, P. Transformers: 'The End of History' for Natural Language Processing? In *Machine Learning and Knowledge Discovery in Databases*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2021; pp. 677–693. [CrossRef]
37. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics Findings of ACL: EMNLP 2020*; Association for Computational Linguistics: Kerrville, TX, USA, 2020; pp. 1536–1547.
38. Zhou, X.; Han, D.; Lo, D. Assessing Generalizability of CodeBERT. In Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), Luxembourg, 27 September–1 October 2021. [CrossRef]
39. Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; Liu, P.J. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.* **2020**, *21*, 1–67. Available online: http://jmlr.org/papers/v21/20-074.html (accessed on 24 December 2023).
40. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 1877–1901.
41. Yang, Z.; Dai, Z.; Yang, Y.; Carbonell, J.; Salakhutdinov, R.; Le, Q.V. XLNet: Generalized autoregressive pretraining for language understanding. *Adv. Neural Inf. Process. Syst.* **2019**, *32*, 5753–5763.
42. Zhang, F.; Yu, X.; Keung, J.; Li, F.; Xie, Z.; Yang, Z.; Ma, C.; Zhang, Z. Improving Stack Overflow question title generation with copying enhanced CodeBERT model and bi-modal information. *Inf. Softw. Technol.* **2022**, *148*, 106922. [CrossRef]
43. Liu, K.; Yang, G.; Chen, X.; Zhou, Y. EL-CodeBert: Better Exploiting CodeBert to Support Source Code-Related Classification Tasks. In Proceedings of the 13th Asia-Pacific Symposium on Internetware, Hohhot, China, 11–12 June 2022. [CrossRef]

44. Wang, R.; Zhang, H.; Lu, G.; Lyu, L.; Lyu, C. Fret: Functional Reinforced Transformer with BERT for Code Summarization. *IEEE Access* **2020**, *8*, 135591–135604. [CrossRef]
45. Yang, Z.; Keung, J.; Yu, X.; Gu, X.; Wei, Z.; Ma, X.; Zhang, M. A Multi-Modal Transformer-based Code Summarization Approach for Smart Contracts. In Proceedings of the 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), Madrid, Spain, 20–21 May 2021. [CrossRef]
46. Hou, S.; Chen, L.; Ye, Y. Summarizing Source Code from Structure and Context. In Proceedings of the 2022 International Joint Conference on Neural Networks (IJCNN), Padua, Italy, 18–23 July 2022. [CrossRef]
47. Wang, Y.; Dong, Y.; Lu, X.; Zhou, A. GypSum: Learning hybrid representations for code summarization. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, Online, 16–17 May 2022. [CrossRef]
48. Gu, J.; Salza, P.; Gall, H.C. Assemble Foundation Models for Automatic Code Summarization. In Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 15–18 March 2022. [CrossRef]
49. Ma, Z.; Gao, Y.; Lyu, L.; Lyu, C. MMF3: Neural Code Summarization Based on Multi-Modal Fine-Grained Feature Fusion. In Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Helsinki, Finland, 29–23 September 2022. [CrossRef]
50. Gao, Y.; Lyu, C. M2TS: Multi-scale multi-modal approach based on transformer for source code summarization. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, Online, 16–17 May 2022. [CrossRef]
51. Ferretti, C.; Saletta, M. Naturalness in Source Code Summarization. How Significant is it? In Proceedings of the 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC), Melbourne, VI, Australia, 15–16 May 2023. [CrossRef]
52. Choi, Y.; Na, C.; Kim, H.; Lee, J.-H. READSUM: Retrieval-Augmented Adaptive Transformer for Source Code Summarization. *IEEE Access* **2023**, *11*, 51155–51165. [CrossRef]
53. Aladics, T.; Jasz, J.; Ferenc, R. Bug Prediction Using Source Code Embedding Based on Doc2Vec. In *Computational Science and Its Applications*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2021; pp. 382–397. [CrossRef]
54. Cheng, X.; Zhang, G.; Wang, H.; Sui, Y. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Online, Republic of Korea, 18–22 July 2022. [CrossRef]
55. Hegedus, P.; Ferenc, R. Static Code Analysis Alarms Filtering Reloaded: A New Real-World Dataset and its ML-Based Utilization. *IEEE Access* **2022**, *10*, 55090–55101. [CrossRef]
56. Bagheri, A.; Hegedus, P. A Comparison of Different Source Code Representation Methods for Vulnerability Prediction in Python. In *Quality of Information and Communications Technology*; Springer: Cham, Switzerland, 2021; pp. 267–281. [CrossRef]
57. Gomes, L.; da Silva Torres, R.; Cortes, M.L. BERT- and TF-IDF-based feature extraction for long-lived bug prediction in FLOSS: A comparative study. *Inf. Softw. Technol.* **2023**, *160*, 107217. [CrossRef]
58. Pan, C.; Lu, M.; Xu, B. An Empirical Study on Software Defect Prediction Using CodeBERT Model. *Appl. Sci.* **2021**, *11*, 4793. [CrossRef]
59. Ma, X.; Keung, J.W.; Yu, X.; Zou, H.; Zhang, J.; Li, Y. AttSum: A Deep Attention-Based Summarization Model for Bug Report Title Generation. *IEEE Trans. Reliab.* **2023**, *72*, 1663–1677. [CrossRef]
60. Mahbub, P.; Shuvo, O.; Rahman, M.M. Explaining Software Bugs Leveraging Code Structures in Neural Machine Translation. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, VI, Australia, 14–20 May 2023. [CrossRef]
61. Csuvik, V.; Horvath, D.; Lajko, M.; Vidacs, L. Exploring Plausible Patches Using Source Code Embeddings in JavaScript. In Proceedings of the 2021 IEEE/ACM International Workshop on Automated Program Repair (APR), Madrid, Spain, 1 June 2021. [CrossRef]
62. Mashhadi, E.; Hemmati, H. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. In Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 17–19 May 2021. [CrossRef]
63. Chakraborty, S.; Ray, B. On Multi-Modal Learning of Editing Source Code. In Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, VI, Australia, 15–19 November 2021. [CrossRef]
64. Lajko, M.; Csuvik, V.; Vidacs, L. Towards JavaScript program repair with generative pre-trained transformer (GPT-2). In Proceedings of the Third International Workshop on Automated Program Repair, Pittsburgh, PA, USA, 19 May 2022. [CrossRef]
65. Chi, J.; Qu, Y.; Liu, T.; Zheng, Q.; Yin, H. SeqTrans: Automatic Vulnerability Fix Via Sequence to Sequence Learning. *IEEE Trans. Softw. Eng.* **2023**, *49*, 564–585. [CrossRef]
66. Chen, Z.; Kommrusch, S.; Monperrus, M. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE Trans. Softw. Eng.* **2023**, *49*, 147–165. [CrossRef]
67. Kim, T.; Yang, G. Predicting Duplicate in Bug Report Using Topic-Based Duplicate Learning with Fine Tuning-Based BERT Algorithm. *IEEE Access* **2022**, *10*, 129666–129675. [CrossRef]
68. Dinella, E.; Ryan, G.; Mytkowicz, T.; Lahiri, S.K. TOGA: A neural method for test oracle generation. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 21–29 May 2022. [CrossRef]
69. da Silva, A.F.; Borin, E.; Pereira, F.M.Q.; Queiroz, N.L.; Napoli, O.O. Program representations for predictive compilation: State of affairs in the early 20's. *J. Comput. Lang.* **2022**, *73*, 101171. [CrossRef]

70. Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; Stoyanov, V. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv* **2019**, arXiv:1907.11692.
71. Dai, Z.; Yang, Z.; Yang, Y.; Carbonell, J.; Le, Q.V.; Salakhutdinov, R. Transformer-XL: Attentive language models beyond a fixed-length context. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, Florence, Italy, 28 July–2 August 2019; pp. 2978–2988.
72. Izadi, M.; Gismondi, R.; Gousios, G. CodeFill: Multi-token code completion by jointly learning from structure and naming sequences. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 21–29 May 2022. [CrossRef]
73. Liu, F.; Li, G.; Zhao, Y.; Jin, Z. Multi-task learning based pre-trained language model for code completion. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Virtual Event Australia, 21–25 December 2020. [CrossRef]
74. Lewis, M.; Liu, Y.; Goyal, N.; Ghazvininejad, M.; Mohamed, A.; Levy, O.; Stoyanov, V.; Zettlemoyer, L. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In Proceedings of the Annual Meeting of the Association for Computational Linguistics, Online, 5–10 July 2020; pp. 7871–7880.
75. Kim, S.; Zhao, J.; Tian, Y.; Chandra, S. Code Prediction by Feeding Trees to Transformers. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, Spania, 22–30 May 2021. [CrossRef]
76. Gemmell, C.; Rossetto, F.; Dalton, J. Relevance Transformer: Generating Concise Code Snippets with Relevance Feedback. In Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, Virtual Event China, 25–30 July 2020. [CrossRef]
77. Soliman, A.S.; Hadhoud, M.M.; Shaheen, S.I. MarianCG: A code generation transformer model inspired by machine translation. *J. Eng. Appl. Sci.* **2022**, *69*, 104. [CrossRef]
78. Yang, G.; Zhou, Y.; Chen, X.; Zhang, X.; Han, T.; Chen, T. ExploitGen: Template-augmented exploit code generation based on CodeBERT. *J. Syst. Softw.* **2023**, *197*, 111577. [CrossRef]
79. Laskari, N.K.; Reddy, K.A.N.; Indrasena Reddy, M. Seq2Code: Transformer-Based Encoder-Decoder Model for Python Source Code Generation. In *Third Congress on Intelligent Systems*; Lecture Notes in Networks and Systems; Springer: Singapore, 2023; pp. 301–309. [CrossRef]
80. Bui, N.D.Q.; Yu, Y.; Jiang, L. Bilateral Dependency Neural Networks for Cross-Language Algorithm Classification. In Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24–27 February 2019. [CrossRef]
81. Yang, G.; Zhou, Y.; Chen, X.; Yu, C. Fine-grained Pseudo-code Generation Method via Code Feature Extraction and Transformer. In Proceedings of the 2021 28th Asia-Pacific Software Engineering Conference (APSEC), Taipei, Taiwan, 6–9 December 2021. [CrossRef]
82. Alokla, A.; Gad, W.; Nazih, W.; Aref, M.; Salem, A.-B. Retrieval-Based Transformer Pseudocode Generation. *Mathematics* **2022**, *10*, 604. [CrossRef]
83. Gad, W.; Alokla, A.; Nazih, W.; Aref, M.; Salem, A. DLBT: Deep Learning-Based Transformer to Generate Pseudo-Code from Source Code. *Comput. Mater. Contin.* **2022**, *70*, 3117–3132. [CrossRef]
84. Acharjee, U.K.; Arefin, M.; Hossen, K.M.; Uddin, M.N.; Uddin, M.A.; Islam, L. Sequence-to-Sequence Learning-Based Conversion of Pseudo-Code to Source Code Using Neural Translation Approach. *IEEE Access* **2022**, *10*, 26730–26742. [CrossRef]
85. Shahbazi, R.; Sharma, R.; Fard, F.H. API2Com: On the Improvement of Automatically Generated Code Comments Using API Documentations. In Proceedings of the 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), Madrid, Spain, 20–21 May 2021. [CrossRef]
86. Yang, G.; Chen, X.; Cao, J.; Xu, S.; Cui, Z.; Yu, C.; Liu, K. ComFormer: Code Comment Generation via Transformer and Fusion Method-based Hybrid Code Representation. In Proceedings of the 2021 8th International Conference on Dependable Systems and Their Applications (DSA), Yinchuan, China, 5–6 August 2021. [CrossRef]
87. Chakraborty, S.; Ahmed, T.; Ding, Y.; Devanbu, P.T.; Ray, B. NatGen: Generative pre-training by "naturalizing" source code. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 14–18 November 2022. [CrossRef]
88. Geng, M.; Wang, S.; Dong, D.; Wang, H.; Cao, S.; Zhang, K.; Jin, Z. Interpretation-based Code Summarization. In Proceedings of the 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC), Melbourne, VI, Australia, 15–16 May 2023. [CrossRef]
89. Thongtanunam, P.; Pornprasit, C.; Tantithamthavorn, C. AutoTransform: Automated code transformation to support modern code review process. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 21–29 May 2022. [CrossRef]
90. Yu, C.; Yang, G.; Chen, X.; Liu, K.; Zhou, Y. BashExplainer: Retrieval-Augmented Bash Code Comment Generation based on Fine-tuned CodeBERT. In Proceeding of the 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), Limassol, Cyprus, 3–7 October 2022. [CrossRef]
91. Lin, B.; Wang, S.; Liu, Z.; Xia, X.; Mao, X. Predictive Comment Updating with Heuristics and AST-Path-Based Neural Learning: A Two-Phase Approach. *IEEE Trans. Softw. Eng.* **2023**, *49*, 1640–1660. [CrossRef]

92. Karakatic, S.; MiloÅ¡evic, A.; Hericko, T. Software system comparison with semantic source code embeddings. *Empir. Softw. Eng.* **2022**, *27*, 70. [CrossRef]
93. Siddiq, M.L.; Majumder, S.H.; Mim, M.R.; Jajodia, S.; Santos, J.C.S. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In Proceedings of the 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM), Limassol, Cyprus, 3 October 2022. [CrossRef]
94. Yu, L.; Lu, Y.; Shen, Y.; Huang, H.; Zhu, K. BEDetector: A Two-Channel Encoding Method to Detect Vulnerabilities Based on Binary Similarity. *IEEE Access* **2021**, *9*, 51631–51645. [CrossRef]
95. Mateless, R.; Tsur, O.; Moskovitch, R. Pkg2Vec: Hierarchical package embedding for code authorship attribution. *Future Gener. Comput. Syst.* **2021**, *116*, 49–60. [CrossRef]
96. Arshad, S.; Abid, S.; Shamail, S. CodeBERT for Code Clone Detection: A Replication Study. In Proceedings of the 2022 IEEE 16th International Workshop on Software Clones (IWSC), Limassol, Cyprus, 2 October 2022. [CrossRef]
97. Kovacevic, A.; Slivka, J.; Vidakovic, D.; Grujic, K.-G.; Luburic, N.; Prokic, S.; Sladic, G. Automatic detection of Long Method and God Class code smells through neural source code embeddings. *Expert Syst. Appl.* **2022**, *204*, 117607. [CrossRef]
98. Zhang, A.; Fang, L.; Ge, C.; Li, P.; Liu, Z. Efficient transformer with code token learner for code clone detection. *J. Syst. Softw.* **2023**, *197*, 111557. [CrossRef]
99. Liu, K.; Kim, D.; Bissyande, T.F.; Kim, T.; Kim, K.; Koyuncu, A.; Kim, S.; Le Traon, Y. Learning to Spot and Refactor Inconsistent Method Names. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019. [CrossRef]
100. Cabrera Lozoya, R.; Baumann, A.; Sabetta, A.; Bezzi, M. Commit2Vec: Learning Distributed Representations of Code Changes. *SN Comput. Sci.* **2021**, *2*, 150. [CrossRef]
101. Wang, S.; Wen, M.; Lin, B.; Mao, X. Lightweight global and local contexts guided method name recommendation with prior knowledge. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, 23–28 August 2021. [CrossRef]
102. Nguyen, S.; Phan, H.; Le, T.; Nguyen, T.N. Suggesting natural method names to check name consistencies. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20). Association for Computing Machinery, New York, NY, USA; 2020; pp. 1372–1384. [CrossRef]
103. Xie, R.; Chen, L.; Ye, W.; Li, Z.; Hu, T.; Du, D.; Zhang, S. DeepLink: A Code Knowledge Graph Based Deep Learning Approach for Issue-Commit Link Recovery. In Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24–27 February 2019. [CrossRef]
104. Borovits, N.; Kumara, I.; Krishnan, P.; Palma, S.D.; Di Nucci, D.; Palomba, F.; Tamburri, D.A.; van den Heuvel, W.-J. DeepIaC: Deep learning-based linguistic anti-pattern detection in IaC. In Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation, Virtual, USA, 13 November 2020. [CrossRef]
105. Ma, W.; Zhao, M.; Soremekun, E.; Hu, Q.; Zhang, J.M.; Papadakis, M.; Cordy, M.; Xie, X.; Traon, Y.L. GraphCode2Vec: Generic code embedding via lexical and program dependence analysis. In Proceedings of the 19th International Conference on Mining Software Repositories, Pittsburg, PA, USA, 23–24 May 2022. [CrossRef]
106. Wan, Y.; He, Y.; Bi, Z.; Zhang, J.; Sui, Y.; Zhang, H.; Hashimoto, K.; Jin, H.; Xu, G.; Xiong, C.; et al. NaturalCC: An Open-Source Toolkit for Code Intelligence. In Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Pittsburgh, PA, USA, 22–24 May 2022. [CrossRef]
107. Zaharia, S.; Rebedea, T.; Trausan-Matu, S. CWE Pattern Identification using Semantical Clustering of Programming Language Keywords. In Proceedings of the 2021 23rd International Conference on Control Systems and Computer Science (CSCS), Bucharest, Romania, 26–28 May 2021. [CrossRef]
108. Zaharia, S.; Rebedea, T.; Trausan-Matu, S. Machine Learning-Based Security Pattern Recognition Techniques for Code Developers. *Appl. Sci.* **2022**, *12*, 12463. [CrossRef]
109. Barr, J.R.; Shaw, P.; Abu-Khzam, F.N.; Thatcher, T.; Yu, S. Vulnerability Rating of Source Code with Token Embedding and Combinatorial Algorithms. *Int. J. Semant. Comput.* **2020**, *14*, 501–516. [CrossRef]
110. Saletta, M.; Ferretti, C. A Neural Embedding for Source Code: Security Analysis and CWE Lists. In Proceedings of the 2020 IEEE International Conference on Dependable, Autonomic and Secure Computing, International Conference on Pervasive Intelligence and Computing, International Conference on Cloud and Big Data Computing, International Conference on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), Calgary, AB, Canada, 17–22 August 2020. [CrossRef]
111. Hamed, A.A.; Zachara-Szymanska, M.; Wu, X. Safeguarding authenticity for mitigating the harms of generative AI: Issues, research agenda, and policies for detection, fact-checking, and ethical AI. *IScience* **2024**, *27*, 108782. [CrossRef]